Fault Diversity among Off-The-Shelf SQL Database Servers

Ilir Gashi, Peter Popov, Lorenzo Strigini
Centre for Software Reliability,
City University,
Northampton Square
London, EC1V 0HB
i.gashi@city.ac.uk, ptp@csr.city.ac.uk, strigini@csr.city.ac.uk

Abstract

Fault tolerance is often the only viable way of obtaining the required system dependability from systems built out of "off-the-shelf" (OTS) products. We have studied a sample of bug reports from four off-the-shelf SQL servers so as to estimate the possible advantages of software fault tolerance - in the form of modular redundancy with diversity - in complex off-the-shelf software. We checked whether these bugs would cause coincident failures in more than one of the servers. We found that very few bugs affected two of the four servers, and none caused failures in more than two. We also found that only four of these bugs would cause identical, undetectable failures in two servers. Therefore, a fault-tolerant server, built with diverse off-the-shelf servers, seems to have a good chance of delivering improvements in availability and failure rates compared with the individual off-the-shelf servers or their replicated, non-diverse configurations.

1. Introduction

When systems are built out of "off-the shelf" (OTS) products, fault tolerance is often the only viable way of obtaining the required system dependability [23, 30, 12]. Fault tolerance may take multiple forms, from simple error detection and recovery add-ons (e.g. wrappers [22]) to full-fledged "diverse modular redundancy" [16]: replication with diverse versions of the components. Even this latter class of solutions becomes affordable with many OTS products and has the advantage of a fairly simple architecture. The cost of procuring two or even more OTS products (some of which may be free) would still be far less than that of developing one's own.

All these design solutions are well known from the literature. The questions, for the developers of a system using OTS components, are about the dependability gains, implementation difficulties and extra cost that they would bring for that specific system.

To study the issues for a realistic category of OTS products we have chosen SQL database servers. These are complex products, with many faults in each release, and even features that imply an accepted possibility of an incorrect behaviour, albeit rare. An example of the latter is the known "write skew" [3] problem with some optimistic concurrency control architectures [7]. Further dependability improvement of OTS SQL servers seems only possible if fault tolerance through design diversity is used [11]. Given the many available OTS SQL servers and the standardisation of their functionality (SQL 92 and SQL 99), it seems reasonable to build a fault-tolerant SQL server from available OTS servers.

The effort of developing an SQL server using design diversity (e.g. several of-the-shelf SQL servers and suitably adapted "middleware" for replication management) would require strong evidence of its usefulness: for example empirical evidence that likely failures of the SQL servers, which may lead to serious consequences, are unlikely to be tolerated without diversity. This paper starts to investigate such empirical evidence. We seek to demonstrate whether design diversity has a potential to deliver significant improvement of dependability of SQL servers, compared to solutions for data replication that can only tolerate crash failures. To this aim we are running experiments to determine the dependability gains achieved through fault tolerance.

A preliminary evaluation step concerns *fault* diversity rather than *failure* diversity. By manual selection of test cases, one can check whether the diverse redundant configuration would tolerate the known bugs in the repositories of bugs reported for the various OTS servers. We have conducted a study on four SQL servers, both commercial and open-source. We collected known bug reports for these servers. For each bug, we took the test case that would trigger it and ran it on all four servers (if possible), to check for coincident failures. We found the number of coincident failures to be very low.

We use the following terminology. The known bugs for the OTS servers are documented in bug report repositories (i.e. bug databases, mailing lists etc). Each *bug report* contains the description of what the bug is and the *bug script* (SQL code that contains the failure triggering conditions) required to reproduce the *failure* (the erroneous output that the reporter of the bug observed). In our study we collected these bug reports and *ran* the bug scripts in the servers (we will use the phrase "*running a bug*" for the sake of brevity).

This paper is structured as follows: In Section 2 we describe the background and motivation of the study and related work from the literature. In Section 3 we describe how the study was conducted and the terminology for classification of faults. In Section 4 we present the quantitative results obtained. In Section 5 we describe the bugs that caused coincident failures. In Section 6 we discuss the possible reliability gains to be had from using diverse OTS SQL servers and in Section 7 we present conclusions and possible further work.

2. Background and related work

2.1. Fault tolerance in databases

Software fault tolerance has been thoroughly studied and successfully applied in many sectors, including databases. For example, standard database mechanisms such as transaction "rollback and retry" and "checkpointing" can be used to tolerate faults that are due to transient conditions. These techniques can be used with or without data replication in the databases.

There are many solutions for data replication [4, 33, 20], as a feature of many commercial SQL servers or as middleware that can be used with a variety of SQL servers. Typically, these replication solutions work with sets of identical servers. Jimenez-Peris et al [13] present a relevant discussion of the various ways in which database replication with OTS servers can be organised, namely treating the servers as white, grey or black boxes. All commercial offerings are of the white-box kind, where code necessary for replication is added inside the server product. The grey-box approach, as implemented in [14], assumes that servers provide specific services to assist with replication. The black-box approach uses the standard interfaces of the servers. Both the grey and black box approaches are implemented via middleware on top of the existing servers. To the best of our knowledge, a common assumption is made in the known replication solutions that the SQL servers will fail in a "fail-stop" manner [26], with detectable clean crashes, and leaving a copy of a correct state for use in recovery. Apart from simplifying the protocols for data replication, the assumption of crash failures also allows for some performance optimisation such as executing the modifying queries on a single server, which then propagates the updates to all other servers involved in the replication, a solution considered adequate by the standardising bodies [28].

These approaches have shortcomings, i.e., they do not protect against failures that are not easily detectable (non-fail-stop), and incorrect updates would be propagated to all the replicas. Using diverse SQL servers instead of servers of the same type would improve error detection and thus reduce the risk from incorrect results. Availability could also be improved because servers that are diagnosed as correct can continue operation while recovery is performed on the faulty server[s]. Elsewhere [21, 9] we describe some initial steps toward implementing middleware for data replication with diverse SQL servers. There, we also discuss some difficulties of data replication with diverse servers, such as the need to use the subset of SQL that is common to all servers used, and to translate all queries into the SQL "dialects" of these servers.

2.2. Studies of faults and failures

The usefulness of diversity depends on the frequency of those failures that cannot be tolerated without it. There have been comparatively few related studies.

Gray studied the TANDEM NonStop system [10] and observed that over an (unspecified) measured period only one out of 132 faults caused failures deterministically, i.e. the same failure was observed on retry. Gray calls these "Bohrbugs". The others, which he calls "Heisenbugs" only caused failures under special conditions (e.g. created by a combination of the state of the operating system and other software), difficult to reproduce artificially. Heisenbugs – so long as their failures are detected – can be tolerated by replication without diversity, as in the Tandem system. A later study, [17] of field software failures for the Tandem Guardian90 operating system found that 82 % of the reported field software faults were tolerated. However, 18 % of the faults did lead to both non-diverse processes in a Tandem process failing and therefore leading to a system failure.

Related studies exist on determinism and fail-stop properties of database failures, but they, like our study, concern faults rather than failure measurements. A study [5] examined fault reports of three applications (Apache Web server, GNOME and MySQL server). Only a small fraction of the faults (5-14%) were Heisenbugs triggered by transient conditions that would be tolerated by a simple "rollback and retry" approach. However the reason why there are few Heisenbugs here, and indeed in our study, might be that people are less likely to report faults that they cannot reproduce, and this is acknowledged by the authors in [5]. In another study [6] the same authors found (via fault injection) that a significant number of

faults (7%) violated the fail-stop model by writing incorrect data to stable storage. Even though they report that this number falls to 2% when applying the Postgres95 transaction mechanism, this number still remains high for applications with stringent reliability requirements.

2.3. Diversity with off-the-shelf applications

Other researchers have also considered the potential of diversity for improving the dependability of OTS software. Various architectures have been proposed that use diversity for intrusion tolerance: e.g. HACQIT [25], which demonstrates diverse replication (with two OTS web servers - Microsoft's IIS and Apache web server) to detect failures (especially maliciously caused ones) and initiate recovery; SITAR [32], an intrusion tolerant architecture for distributed services and especially COTS servers; or the Cactus architecture [12], intended to enhance survivability of applications which support diversity among application modules.

In another example, [2] uses diverse Java virtual machines for interoperability rather than for tolerating failures.

3. Description of the study

3.1. Bug reports

Two commercial (Oracle 8.0.5 and Microsoft SQL Server 7 (without any service packs applied)) and two open-source (PostgreSQL Version 7.0.0 and Interbase Version 6.0), SQL servers were used in this study. Interbase, Oracle and MSSQL were all run on the Windows 2000 Professional operating system, whereas PostgreSQL (which is not available for Windows) was run on RedHat Linux 6.0 (Hedwig).

We only used bugs that caused failure of a server's *core engine*. We did not consider other bugs such as those that caused failure to a client application tool or various connectivity API's (JDBC/ODBC etc.), because these functions in a future fault tolerant architecture would be provided by the middleware.

For each of these servers there is an accessible repository of reports of known bugs. We collected: Interbase bugs [27] reported in the period between August 2000 and August 2001; PostgreSQL bugs [24] reported between May 2000 and January 2001; Oracle bugs [19] reported between September 1998 and December 2002. Bug reports for MSSQL [18] do not specify dates; we used all reports for both MSSQL 7 and MSSQL 2000, available as of August 2003, that included "bug scripts" and were core engine bugs. For Oracle and MSSQL we collected reports from longer periods, because for these two servers (both "closed development" servers) some reports do not include bug scripts and we could not check whether the bug was present in other servers. By extending the collection period we obtained reasonably large (though obviously imperfect) samples of bug reports. Despite this, the sample that we could use for Oracle contained only 18 bugs, since most reports omitted the bug scripts.

For each reported bug we attempted to run the corresponding bug script. Full details are available in [8].

3.2. Reproducibility of failures

All these servers offer features that are extensions to the basic SQL standard, and these extensions differ between the servers. Bugs affecting one of these extensions thus literally cannot exist in a server that lacks the extension. We called these "dialect-specific*"* bugs. For example, Interbase bug 217138 [8] uses the UNION operator in views, which PostgreSQL 7.0.0 views do not offer, and thus cannot be run in PostgreSQL: it is a dialect-specific bug.

Another "reproducibility" issue arises when a bug script does not cause failure in the server for which the bug was reported. We called these bugs Heisenbugs, borrowing Gray's terminology [10]. We intend to run the Heisenbugs again in a more stressful simulated environment [21] (with multiple clients and large number of transactions) to see whether repeated trials will give incorrect results.

4. Quantitative results

4.1. Detailed results

In total we included in the study 181 bug reports: 55 for Interbase, 57 for PostgreSQL, 51 for MSSQL and 18 for Oracle. Out of these 181 bugs, 76 were "dialect-specific" (could be run in only one of the four servers); 47 could be run in all four servers; 26 could be run in only two servers and 32 in only three servers.

Each bug was first run on the server for which it was reported, and (after translating the script into the SQL dialect of the respective server) on the other servers. The bugs were classified into dialect-specific and non-dialect-specific bugs; the latter were then further classified into Bohrbugs or Heisenbugs as explained previously. The failures were also classified into different categories according to their effects, as different failure types require different recovery mechanisms:

Engine Crash failures: crashes or halts of the core engine.
Incorrect Result failures: incorrect outputs without engine crashes: the outputs do not conform to the server's specification or to the SQL standard.
Performance failures: correct output, but with an unacceptable time penalty for the particular input.
Other failures.

We also classified the failures according to their detectability by a client of the database servers:
Self-Evident failures: engine crash failures, cases in which the server signals an internal failure as an exception (error message) and performance failures.
Non-Self-Evident failures*:* incorrect result failures, without server exceptions within an accepted time delay.

Table 1 contains the results of this step of the study. Each grey column lists the results produced when the bugs reported for a certain server were run on that server. For example, we collected 55 known Interbase bugs, of which, when run on our installation of the Interbase server, 8 did not cause failures (possible Heisenbugs). The 47 bugs that caused failures are further classified in the part of the column below the double vertical lines, after the "Failure observed" row. All the performance failures and all the engine crashes are self-evident. Incorrect Result failures and "Other" failures can be self-evident or non-self-evident depending on whether the server gives an error message.

The three columns to the right of the grey one present the results of running the Interbase bugs on the other three servers. For example, we can see that 23 of the Interbase bugs cannot be run in PostgreSQL (dialect-specific bugs). Then we have the bugs that "require further work": this means that we have not managed yet to translate the bug script in the PostgreSQL dialect of SQL, or are listed as "performance bugs" but we could not decide whether performance improves by changing servers. We plan to resolve this uncertainty via a testing infrastructure [21] to measure the precise execution times of the queries.

Out of 55 Interbase bugs we managed to run 27 in PostgreSQL; only one caused a failure in both Interbase and PostgreSQL. This particular failure was a non-self-evident incorrect result as can be seen from the table.

As for the failure types, we can see that most of the bugs cause incorrect result failures. This will be discussed further in the Section 6.

We observed a higher number of Heisenbugs in MSSQL and Oracle than in the other servers. This was documented by some of the bug reports, which indicated: "may cause a failure".

Table 1. Results of running the bug scripts on all four servers. *IB* stands for Interbase, *PG* for PostgreSQL, *OR* for Oracle and *MS* for MSSQL

			IB	PG	OR	MS	**PG**	IB	OR	MS	**OR**	IB	MS	PG	**MS**	IB	OR	PG
Total bug scripts			55	55	55	55	57	57	57	57	18	18	18	18	51	51	51	51
Bug script cannot be run (Functionality Missing)			n/a	23	20	16	n/a	32	27	24	n/a	13	13	12	n/a	36	32	31
Further Work			n/a	5	4	6	n/a	2	0	0	n/a	1	1	2	n/a	3	7	2
Total bug scripts run			55	27	31	33	57	23	30	33	18	4	4	4	51	12	12	18
No failure observed			8	26	31	31	5	23	30	31	4	4	4	3	12	11	12	12
Failure observed			47	**1**	0	**2**	52	0	0	**2**	14	0	0	**1**	39	**1**	0	**6**
Types of failures	**Poor Performance**		3	0	0	0	0	0	0	0	1	0	0	0	6	0	0	0
	Engine Crash		7	0	0	0	11	0	0	0	3	0	0	0	5	0	0	0
	Incorrect Result	**Self-evident**	4	0	0	**1**	14	0	0	**1**	3	0	0	0	10	0	0	**6**
		Non-self-evident	23	**1**	0	**1**	20	0	0	**1**	7	0	0	**1**	17	**1**	0	0
	Other	**Self-evident**	2	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0
		Non-self-evident	8	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0

4.2. Summary of observed fault diversity

Table 2 contains a summary from the viewpoint of the probable effects on a fault-tolerant server. Of the 47 bugs that could be run on all four servers, 12 did not cause failures in any of the servers: they are Heisenbugs for the server for which they were reported, and non-existent or Heisenbugs for the other three servers. 31 of these only caused a failure in the server for which they were reported and not in the others; and 4 bugs caused a coincident failure in two servers.

In addition to these 47, we have many bugs that could be run only on a subset of the four servers and thus on a fault-tolerant server built out of this subset. The following sections in the table show the number of bugs that could be run in each of these different combinations (4 three-version combinations and 6 two-version combinations), and how many caused failures or coincident failures.

The last four columns show the 76 dialect-specific bugs, which could only be run in the server for which they were reported and therefore affect functionality that would not be available on a fault-tolerant diverse server.

Table 2. The number of bug scripts run and the effects on different combinations of servers

The server(s) in which the bug script was run	IB, PG, OR, MS	IB, PG, OR only	IB, PG, MS only	IB, OR, MS only	PG, OR, MS only	IB, PG Only	IB, MS Only	IB, OR Only	PG, OR Only	PG, MS Only	MS, OR Only	IB Only	PG Only	MS Only	OR Only
Total number of bug scripts run	47	3	7	12	10	5	3	0	4	12	2	17	18	28	13
Failure not observed in any server	12	0	1	2	0	0	0	0	0	0	1	1	2	5	3
Failure observed in one server only	31	3	6	9	9	5	3	0	3	7	1	16	16	23	10
Failure observed in two servers	**4**	0	0	**1**	**1**	0	0	0	**1**	**5**	0	N/A	N/A	N/A	N/A
None of the bugs caused a failure in more than two servers															

4.3. Two-version combinations

We now look more closely at the two-version combinations of the four different servers in our study, to see how many of the coincident failures are *detectable* in the 2-version systems. We define:

Detectable failures: self-evident failures or those where servers return different incorrect results (the comparison algorithm must be written to allow for possible differences in the representation of correct results, e.g. different numbers of digits in the representation of floating point numbers, padding of characters in character strings etc.). All failures affecting only one out of two (or at most n-1 out of n) versions are detectable.

Non-Detectable failures: the ones for which two (or more) servers return identical incorrect results.

Table 3 contains a summary of the results on each of the six possible two-version combinations. Here we only include bugs that could be run on both servers, i.e. we exclude dialect-specific bugs. Only four of the 12 coincident failures we observed are non-detectable. We can see that diversity allows detection of failures for at least 94% of these bugs.

Table 3. Summary of results for the two-version combinations

Pairs of servers	Total number of bug scripts run	Failure observed (in at least one server)	One out of two servers failing		Both servers failing		
			Self-evident	Non -self-evident	Non – Detectable	Detectable	
						Self-evident	Non-self-evident
IB + PG	62	43	17	25	**1**	0	0
IB + OR	62	29	8	21	0	0	0
IB + MS	69	35	11	21	**2**	**1**	0
PG + OR	64	30	13	16	0	0	**1**
PG + MS	76	46	18	21	**1**	**6**	0
OR + MS	71	14	7	7	0	0	0

5. Common faults

We now discuss the bugs that caused coincident failures, listed in Table 4. We give some details about the functions affected and conjectures about the probable severity and frequency of failure as a function of the environment of use of the server.

There were 13 bugs in total that were originally reported for one server but caused failure in another. *12 caused a failure in both the server for which they were reported and another server*. One bug (MSSQL bug report 56775) was reported for MSSQL, did not cause failure in MSSQL (possible Heisenbug) but did cause failure in PostgreSQL.

Table 4. Bugs that cause coincident failures. The table should be read horizontally to know for which server the bug was reported, and vertically to know in which other server it caused a failure.

	IB	PG	OR	MS
IB	N/A	**1** - (Bug ID 223512)	0	**2**– (BugID's 217042(3), 222476)
PG	0	N/A	0	**2** – (BugID's 43 and 77)
OR	0	**1** – (Bug ID 1059835)	N/A	0
MS	**1-** (BugID 58544)	**5** – (BugID's 54428, 56516, 58158, 58253, 351180)	0	N/A

Arithmetic-related bugs

PostgreSQL bug report 77 and Oracle bug report 1059835 [8] describe arithmetic precision problems, causing incorrect result failures. The Oracle bug 1059835 affects the MOD (modular arithmetic) operator, probably causing higher consequence failures. The failure rates for these bugs would only be expected to be high in applications with high use of mathematical functions, not a typical use of SQL servers.

Bugs affecting complex queries

PostgreSQL bug 43 [8] causes a failure in both PostgreSQL and MSSQL. The complex SELECT statement below, with nested sub-queries, causes the failure:

```
SELECT P.ID AS ID, P.NAME AS NAME FROM PRODUCT
P WHERE P.ID IN
(SELECT ID FROM PRODUCT WHERE PRICE >= '9.00'
AND PRICE <= '50' AND ID NOT IN
((SELECT PRODUCT_ID FROM PRODUCT_SPECIAL
WHERE START_DATE <= '2000-9-6' AND END_DATE >=
'2000-9-6')
UNION
(SELECT PRODUCT_ID AS ID FROM PRODUCT_SPECIAL
WHERE PRICE >= '9.00' AND PRICE <= '50' AND
START_DATE <= '2000-9-6' AND END_DATE >= '2000-9-6')))
```

Interestingly, for this same bug the two servers fail with different patterns. PostgreSQL fails returning a parsing error. MSSQL does not, but subsequently gives an incorrect result, probably because it built an incorrect parsing tree.

MSSQL Bug 58544 [8] causes failures in both MSSQL and Interbase. Using a LEFT OUTER JOIN on a VIEW that uses the DISTINCT keyword causes the failure. A left outer join is a special type of outer join where if you have a join between tables T1 and T2 then the joined table unconditionally has a row for each row in T1 (as opposed to a Full Outer Join where the joined table has a row for each row present in both tables T1 and T2). The DISTINCT keyword subsequently eliminates all the duplicate rows from the joined table. Complex queries would be common on large databases with many tables, leading probably to a comparatively high failure rate, with possibly high failure severity, especially for incorrect result failures.

Miscellaneous bugs

Interbase Bug 223512(2) causes a failure in the Data Definition Language (DDL) part of SQL which is used to create/modify database objects (i.e. tables, views, users, procedures etc). It causes failures in both Interbase and PostgreSQL: both incorrectly allow a client to drop Views using the Drop Table statement. This violates the SQL-92 standard, which allows Views to be dropped only via the Drop View statement. This bug would seem to cause infrequent failures in operation and it would normally require an error by an administrator. The severity of failures would also be expected to be low since a view is just a 'virtual table' (or a stored SELECT statement), which represents the data from one or more tables. No data are lost by dropping a view, although a runtime error will be generated each time a client attempts to access the dropped view.

Interbase bug 217042(3) causes both Interbase and MSSQL to fail to validate the default values upon creation of tables. Therefore a statement like:

```
CREATE TABLE TEST (A INT DEFAULT 'ABC')
```

is allowed in both Interbase and MSSQL, even though an error should be raised since a string value (ABC) cannot be stored in an Integer type attribute. The DEFAULT attributes are used often in operation but it is not clear how often database users will define DEFAULT values of the wrong type. The failure to detect that an incorrect type default value is being assigned to a particular column at table creation time is non-detectable. However, a runtime error will occur, generating an error message, every time an attempt is made to insert the default value into the table: the failure will be detected, albeit with high latency[1].

Interbase bug 222476 causes a failure in MSSQL as well. Both servers give empty field names for *avg* (average) and *sum* SQL functions, although they return correct results in these fields. This would be a serious problem for client applications that construct their output from the field names and results returned by the server.

Five of the MSSQL bug scripts also caused failure in PostgreSQL, but with the difference that PostgreSQL fails at the beginning of the bug script. This implies that the causes are probably different for the two products, and the "failure regions" (sets of demands that would trigger the bug) identified by such scripts for the two servers only partially overlap: there are variations of the script for

[1] If we classify the database as part of the server system, the common terminology recommended in [15] would imply that assigning the wrong type is an internal error, which only becomes a failure and is detected when the attempt is made to insert the default value.

which PostgreSQL fails but MSSQL does not. For example, MSSQL bug 54428 causes an incorrect "primary key constraint" failure in MSSQL. The same bug causes failure (at the beginning of the bug script) when an attempt is made to create a clustered index in PostgreSQL. The latter is a known bug for PostgreSQL, and its correction in the later release 7.0.3 causes PostgreSQL not to fail on any of these five scripts.

6. Discussion

6.1. Extrapolating from the counts of common bugs to reliability of a diverse server

These numbers are intriguing and point to a potential for serious dependability gains from assembling a fault tolerant server from two or more of these off-the-shelf servers. But they are not definitive evidence. Apart from the sampling difficulties caused e.g. by lack of certain bug scripts, it is important to clarify to what extent our observations allow us to predict such gains.

For brevity, we consider the simplest case: suppose that users of a certain database server product A try to obtain a more dependable service by using a fault-tolerant, replicated, diverse server AB, built from product A plus another product B (for discussion of the feasibility and design problems, see [21]). The number of bugs reported over a certain reference period (say one year) for product A is m_A. Our study then finds that of these m_A bugs, only m_{AB} also caused failure of B. We may then expect that, had these users been using AB instead of A, only those failures of A that were due to those m_A bugs could have caused complete service failures. How much more reliable would this have made the AB server, compared to the A server?

Before proceeding, we introduce some more simplifications. The possible effects of individual server failures on system failures have been discussed in Sections 4.1 and 4.3, under the definitions of "self-evident" and "detectable" failures. Here, for the sake of brevity, we use a simplified scenario: failures of both servers A and B on the same demand are "system failures", and failures of a single one of them are not[2]. In addition, we only consider the effects on reliability of the factor that we have studied: the diversity between faults of the two products A and B. We thus ignore any effects of the middleware needed in the AB server, which adds complexity and thus possibly faults; and of added complexity in client applications that used complex vendor-specific features of server A, if they must be adapted to use the more restricted feature set of server AB. With these simplifications, the AB server is certain to be at least as reliable as the single A server because it only fails if both A and B fail. We still need to assess the size of the probable reliability gain. To this end, we need to take into account various complications: the difference between fault records and failure records; imperfect failure reporting; variety of usage profiles.

We can start with a scenario in which our data would be sufficient for trustworthy predictions, and then discuss the effects of these assumptions not holding in practice. This ideal scenario is as follows: we are interested in the reliability gains for a database installation using server A, if it were to switch to a diverse server AB, assuming that this installation has a usage profile (probabilities of all possible demands on the server) similar to the average of all the bug-reporting installations of server A[3]. We assume that users neither change their patterns of usage of the databases (demand profile) nor upgrade to new releases of the database servers[4]; that all failures that affected installations of A during the reference year were noticed and reported; and that there is exactly one bug report for each failure that occurred.

Then, we can state that the bug reports describe a one-year sample of operation of the system, and our best reliability prediction is that the same set of users, during another year of operation, would experience a mean number m_A of system failures if they used A, but only m_{AB} if they used AB. With the numbers we observed, the ratio m_{AB} / m_A is quite small, so the expected reliability gain would be large. Given that the reports come from millions of installations, each submitting many demands[5], we might even trust that the true failure probability per demand is close to the observed frequency of failures.

The first difficulty with this analysis is that reports concern bugs, not how many failures each caused. They

[2] This simplified model is still realistic if either: i) we are only concerned with interruptions of service, and all failures of A and/or B are detectable (crashes, self-detected errors, or different erroneous results if both A and B fail); or ii) we are concerned with undetected erroneous results, and all failures of both A and B on the same demand are pessimistically assumed to produce such results.

[3] Or, from a market-assessment viewpoint, we may consider the average reliability gains for the population of all database installations which depend on server A, if they switched to using AB.

[4] Because we wish to reason about the reliability effects of diversity alone. This scenario also has practical interest, though. Usage patterns vary over time, but periods of very slow variations must exist; users do upgrade to new versions, but upgrades bring expense and new problems, so that it is interesting to see whether diversity would be a more cost-effective way of achieving good average dependability over a system's lifetime than frequent upgrades.

[5] How to define a "demand" to a state-rich system like a database server, for the purpose of inference about reliability, is a tricky theoretical and practical issue. For this informal discussion of other difficulties in inference, we ask the reader to accept that a practical solution can be found, somewhere between a single command and the whole sequence of commands over the lifetime of an installation. (cf e.g. [29]for examples of useful compromises).

do not tell us whether a bug has a large or a small effect on reliability, although the faults that did not cause failures would tend to have stochastically lower effect on reliability than those that caused failures. Thus, the m_{AB} bugs which still cause the fault tolerant server AB to fail may account for a large (perhaps close to 100%) or a small fraction (perhaps close to 0) of the failures observed in A's operation. The actual reliability gain may be anywhere between negligible and very high.

Software is often assessed in terms of number of bugs remaining. But it is easily seen that the bug reports do not give us any information on this number: the m_A bugs reported may be the only bugs in the products, or they may be a fraction of them (perhaps minimal), which happened to be the ones causing failures during the reference year.

Another difficulty is not knowing how many of the failures that occur are actually reported. This fraction is certainly less than 100%. If all failures had the same probability of being reported, the ratio between our predicted failure counts for AB and A would still be the ratio m_{AB} / m_A, although both terms in the ratio would be larger and affected by wider uncertainty. Reporting is probably biased, for instance towards bugs that cause higher frequency or higher severity of failures. Some failures – like crashes – are more noticeable than others, like storing incorrect data in some data fields, which may not produce visible effects for a long time (also making it more difficult to trace the visible problem back to its cause). Some users are more assiduous at producing failure reports, so the bugs that affect them more are also more likely to be reported, even if not so important for other users.

In the end, we do not know in detail how failure reporting differs between different bugs, but bug reports are likely to be better evidence about bugs that cause blatant failures than about subtle (arguably more dangerous) failures. This prompts another consideration: as reported bugs are corrected and products mature, more of their failures are likely to be of the subtler types, unlikely to be reported. Therefore failure underreporting probably causes a bias towards *under*estimating the frequency of failures for which diversity would help. This makes diversity a more attractive defence, but it also means that bug reports will become a less and less accurate representation of the set of failures actually occurring.

Last, we have the problem of usage profiles. A single user organisation needs predictions about the dependability of its specific installation of server AB or A (i.e., with or without diversity), which depends on its specific usage profile, which differs – perhaps by much – from the aggregate profile of the user population which generated the bug reports. Installations that manage different databases, with different user needs, are subjected to different usage profiles. It is then plausible that different bugs are important for different installations; this conjecture is also supported by a possible interpretation of Adams' findings [1] about the surprisingly small average failure rates of many bugs, when averaged over many installations. Then, the number of bugs whose effects can be tolerated (what we have counted here) gives little information about the resulting dependability gains. The actual effect can only be determined empirically. The user organisation may seek indirect evidence from the publicly available bug reports: if they generally match the failures experienced locally, the local effects of tolerating those bugs can be assessed. However if it does not, little insight is gained, and the exercise is time-consuming.

6.2. Decisions about deploying diversity

We have underscored that these results are only *prima facie* evidence for the usefulness of diversity.

A better analysis would be obtained from the actual failure reports (including failure counts), available to the vendors, especially if they use automatic failure reporting mechanisms (users are biased towards under-reporting of failures from bugs they have reported before, or for which they have successful workarounds or recovery mechanisms), and even better if they also have indications about the users' usage profile (from rough measures like the size of the database managed, to detailed monitoring as proposed in [31]). However, vendors are often wary of sharing such detailed dependability information with their customers.

How can then individual user organisations decide whether diversity is a suitable option for them, with their specific requirements and usage profiles? As usual for dependability-enhancing measures, the cost is reasonably easy to assess: costs of the software products, the required middleware, difficulties with client applications that require vendor-specific features, hardware costs, run-time cost of the synchronisation and consistency enforcing mechanisms, and possibly more complex recovery after some failures. The gains in improved reliability and availability (from fewer system failures and easier recovery from some failures, set against possible extra failures due to the added middleware), and possibly less frequent upgrades, are difficult to predict except empirically. This uncertainty will be compounded, for many user organisations, by the lack of trustworthy estimates of their baseline reliability with respect to subtle failures: databases are used with implicit confidence that failures will be self-evident.

We note that for some users the evidence we have presented would already indicate a diverse server to be a reasonable and relatively cheap precautionary choice, even without good predictions of its effects. These are

users who have: serious concerns about dependability (e.g., high costs for interruptions of service or undetected incorrect data being stored); applications which use mostly the core features common to multiple off-the-shelf products (recommended by practitioners to improve portability of the applications); modest throughput requirements for updates, which make it easy to accept the synchronisation delays of a fault-tolerant server.

7. Conclusions

To estimate the possible advantages of modular-redundant diversity in complex off-the-shelf software, we studied a sample of bug reports from four popular off-the-shelf SQL database server products. We checked whether more than one product exhibited bugs that would cause common-mode failures if the products were used in a diverse redundant architecture. It appears that such common bugs are rare. We found very few bugs that affected two of the four servers, and none that affected more than two. Moreover only four of these bugs would cause identical, undetectable failures in two servers. Fault-tolerant, diverse servers seem to have a good chance of improving failure rates and availability.

These preliminary results must be taken with caution, as discussed in Section 6, but are certainly interesting and indicate that this topic deserves further study. Their immediate implications vary between users, but there are classes of database server installations for which even these preliminary results seem to recommend diversity as a prudent and cost-effective strategy. Decisions would of course involve many other considerations which we could not discuss here: performance, total cost of ownership including updates, risks of dependence on one vendor, etc.

The practical obstacle would be the need for "middleware": most users would need an off-the-shelf middleware package, which in turn is not likely to be developed until there are enough users. On the other hand, a dedicated user could develop a middleware package in the hope of seeing his investment amplified through the creation of an open-source community of user/developers. But once the diverse server is running, the dependability changes due to diversity could be directly assessed. The user could decide on an ongoing basis which architecture is giving the best trade-off between performance and dependability, from a single server to the most pessimistic fault-tolerant configuration (with tight synchronisation and comparison of results at each query).

Some other interesting observations include:

- it may be worthwhile for vendors to test their servers using the known bug reports for other servers. For example, we observed 4 MSSQL bugs that had not been reported in the MSSQL service packs (previous to our observation period). Oracle was the only server that never failed when running on it the reported bugs of the other servers;
- the majority of bugs reported, for all servers, led to "incorrect result" failures (64.5%) rather than crashes (17.1%) (despite crashes being more obvious to the user). This is contrary to the common assumption that the majority of bugs lead to an engine crash, and warrants more attention by users to fault-tolerant solutions, and by designers of fault-tolerant solutions to tolerating subtle and non fail-silent failures.

Future work that is desirable includes:

- repeating this study on later releases of the servers, to verify whether the general conclusions drawn here are repeated, indicating that they are the consequences of factors that do not disappear with the evolution of the software products;
- statistical testing to assess the actual reliability gains. This is already under way. We have run a few million queries with various loads including experiments based on the TPC-C benchmark. We have not observed any failures so far (however, with the TPC-C load we found that a significant gain in performance can be obtained with diverse servers [9]). We plan to continue these experiments with more complete test loads. These are important for their own sake, as evidence for decision-making, but also for the side benefit of checking how far the data confirm the impressions gained from this study, and thus how accurate a picture fault reports paint for these products;
- studying alternative options for software fault tolerance with OTS servers, e.g. wrappers rephrasing queries into alternative, logically equivalent sets of statements to be sent to replicated, even non-diverse servers [9];
- developing the necessary components for users to be able to try out diversity in their own installations, since the main obstacle now is the lack of popular off-the-shelf "middleware" packages for data replication with diverse SQL servers.

Acknowledgment

This work was supported in part by the "Diversity with Off-The-Shelf components" (DOTS) Project funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC). We would also like to thank Bev Littlewood, Peter Bishop and the anonymous DSN reviewers for comments on an earlier version of this paper.

References

[1] E. N. Adams, "Optimizing preventive service of software products", IBM Journal of Research and Development, 28, 1984, pp. 2-14.

[2] C. Allaire, "Allaire Run: Edition comparison", www.allaire.com/products/jrun/MoreInformation/ChoosingTheEdition.cfm.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil, "A Critique of ANSI SQL Isolation Levels", in Proc. SIGMOD Int. Conf. on Management of Data, 1995.

[4] P. A. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems", Reading, Mass., Addison-Wesley, 1987.

[5] S. Chandra and P.M. Chen, "Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software", in Proc. IEEE DSN 2000, NY, USA, 2000, pp. 97-106.

[6] S. Chandra and P.M. Chen, "How fail-stop are programs", in Proc. IEEE FTCS-28, Munich, Germany,1998, pp. 240-249.

[7] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha, "Making Snapshots Isolation Serializable", 2000, http://www.cs.umb.edu/~isotest/snaptest/snaptest.pdf.

[8] I. Gashi, "Tables containing known bug scripts of Interbase, PostgreSQL, Oracle and MSSQL." 2003, http://www.csr.city.ac.uk/people/ilir.gashi/DSN/.

[9] I. Gashi, P. Popov, V. Stankovic and L. Strigini, "On designing dependable services with diverse off-the-shelf SQL servers", in A. Romanovsky, R. de Lemos and C. Gacek (Ed.) "Architecting Dependable Systems", Springer, 2004: in print.

[10] J. Gray, "Why do computers stop and what can be done about it?" in Proc. 6th International Conference on Reliability and Distributed Databases, 1987.

[11] J. Gray, "FT101: Talk at UC Berkeley on Fault-Tolerance",2000,http://research.microsoft.com/~Gray/talks/UCBerkeley_Gray_FT_Avialiability_talk.ppt.

[12] M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte and G. T. Wong, "Survivability through Customization and Adaptability: The Cactus Approach", in Proc. DARPA Information Survivability Conference & Exposition, 2000.

[13] R. Jimenez-Peris and M. Patino-Martinez, "D5: Transaction Support", ADAPT Middleware Technologies for Adaptive and Composable Distributed Components Deliverable IST-2001-37126, 2003,
http://adapt.ls.fi.upm.es/deliverables/transactions.pdf.

[14] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso and B. Kemme, "Scalable Database Replication Middleware", in Proc. 22nd IEEE Int. Conf. on Distributed Computing Systems, Vienna, Austria, 2002, pp. 477-484.

[15] J. C. Laprie (Ed.), "Dependability: Basic Concepts and Associated Terminology", Springer-Verlag, 1991.

[16] J. C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures", IEEE Computer, 23, 1990, pp. 39-51.

[17] I. Lee and R. K. Iyer, "Faults, Symptoms and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System", in Proc. IEEE FTCS-23, Toulouse, France, 1993, pp. 20-29.

[18] Microsoft, "List of Bugs Fixed by SQL Server 7.0 Service Packs",http://support.microsoft.com/default.aspx?scid=kb;EN=US;313980.

[19] Oracle, "Oracle Metalink",
http://metalink.oracle.com/metalink/plsql/ml2_gui.startup.

[20] F. Pedone and S. Frolund, "Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases", in Proc. 19th IEEE Symp. on Reliable Distributed Systems (SRDS'00), Nurnberg, Germany, 2000, pp. 176-185.

[21] P. Popov, L. Strigini, A. Kostov, V. Mollov and D. Selensky, "Software Fault-Tolerance with Off-the-Shelf SQL Servers", in Proc. 3rd Int. Conf. on COTS-based Software Systems, ICCBSS'04, Redondo Beach, CA USA, 2004: in print.

[22] P. Popov, L. Strigini, S. Riddle and A. Romanovsky, "Protective Wrapping of OTS Components", in Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto, 2001.

[23] P. Popov, L. Strigini and A. Romanovsky, "Diversity for off-the-Shelf Components", in Proc. IEEE DSN 2000, - Fast Abstracts supplement, New York, USA, 2000, pp. B60-B61.

[24] PostgreSQL, "PostgreSQL Bugs mailing list archives", http://archives.postgresql.org/pgsql-bugs/.

[25] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich and K. Levitt, "The Design and Implementation of an Intrusion Tolerant System", in Proc. IEEE DSN 2002, Washington, USA, 2002, pp. 285-292.

[26] F. B. Schneider, "Byzantine generals in action: Implementing fail-stop processors", ACM TOCS, 2(2), 1984, pp. 145-154.

[27] SourceForge, "Interbase (Firebird) Bug tracker", http://sourceforge.net/tracker/?atid=109028&group_id=9028&func=browse.

[28] H. Sutter, "SQL/Replication Scope and Requirements document", ISO/IEC JTC 1/SC 32 Data Management and Interchange WG3 Database Languages, H2-2000-568, 2000

[29] J. Tian, L. Peng and J. Palma, "Test-execution-based reliability measurement and modeling for large commercial software", IEEE TSE, 21, 1995, pp. 405-414.

[30] A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saidi, V. Stavridou and T. E. Uribe, "An Adaptive Intrusion-Tolerant Server Architecture", 1999,
http://www.sdl.sri.com/users/valdes/DIT_arch.pdf.

[31] J. Voas, "Deriving Accurate Operational Profiles for Mass-MarketedSoftware",
http://www.cigital.com/papers/download/profile.pdf.

[32] F. Wang, F. Gong, C. Sargor, K. Goseva-Popstojanova, K. Trivedi and F. Jou, "SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services", in Proc. IEEE Workshop on Information Assurance and Security, West Point, NY, U.S.A, 2001.

[33] M. Weismann, F. Pedone and A. Schiper, "Database Replication Techniques: a Three Parameter Classification", in Proc. 19th IEEE Symp. on Reliable Distributed Systems (SRDS'00), Nurnberg, Germany, 2000, pp. 206-217.